Load Imbalance in Parallel Programs [*]

Maria Calzarossa, Luisa Massari, and Daniele Tessera

Dipartimento di Informatica e Sistemistica,
Università di Pavia,
I-27100 Pavia, Italy,
{mcc,massari,tessera}@unipv.it

Abstract. Parallel programs experience performance inefficiencies as a result of dependencies, resource contentions, uneven work distributions and loss of synchronizations among processors. The analysis of these inefficiencies is very important for tuning and performance debugging studies. In this paper we address the identification and localization of performance inefficiencies from a methodological viewpoint. We follow a top down approach. We first analyze the performance properties of the programs at a coarse grain. We then study the behavior of the processors and their load imbalance. The methodology is illustrated on a study of a message passing computational fluid dynamic program.

1 Introduction

The performance achieved by a parallel program is the result of complex interactions between the hardware and software resources involved in its execution. The characteristics of the program, that is, its algorithmic structure and input parameters, determine how it can exploit the available resources and the allocated processors. Hence, tuning and performance debugging of parallel programs are challenging issues [11].

Tuning and performance debugging typically rely on an experimental approach based on instrumenting the program, monitoring its execution and analyzing the performance measures either on the fly or post mortem. Many tools have been developed for this purpose (see e.g., [1], [2], [5], [12], [13], [14]). These tools analyze the behavior of the various activities of a program, e.g., computation, communication, synchronization, by means of visualization and statistical analysis techniques. Their major drawback is that they fail to assist users in mastering the complexity inherent in the analysis of parallel programs.

Few tools focus on the analysis of parallel programs with the aim of identifying their performance bottlenecks, that is, the code regions critical from the performance viewpoint. The Poirot project [6] proposed a tool architecture to automatically diagnose parallel programs using a heuristic classification scheme.

[*] This work has been supported by the Italian Ministry of Education, University and Research (MIUR) under the FIRB Programme, by the University of Pavia under the FAR Programme and by the Italian Research Council (CNR).

The Paradyn Parallel Performance tool [9] dynamically instruments the programs to automate bottleneck detection during their execution. The Paradyn Performance Consultant starts a hierarchical search of the bottlenecks, defined as the code regions of the program whose performance metrics exceed some predefined thresholds. The automated search performs a stack sampling [10] and a pruning of the search space based on historical performance and structural data [7].

In this paper we address the analysis of the performance inefficiencies of parallel programs from a methodological viewpoint. We study the behavior and the performance properties of the programs with the aim of detecting the symptoms of performance problems and localizing where they occurred. Our methodology is based on the definition of performance metrics and on the use of a few criteria able to explain the performance properties of the programs and the inefficiencies due to load imbalance among the processors.

The paper is organized as follows. Section 2 introduces the metrics and criteria for the evaluation of the overall behavior of parallel programs. Section 3 focuses on the analysis of the behavior of the allocated processors. An application of the methodology is presented in Section 4. Finally, Section 5 concludes the paper and outlines guidelines towards the integration of our methodology into a performance analysis tool.

2 Performance Properties

Tuning and debugging the performance of a parallel program can be seen as an iterative process consisting of several steps, dealing with the identification and localization of inefficiencies, their repair and the verification and validation of the achieved performance.

As already stated, our objective is to address the performance analysis process by focusing on the identification and localization of performance inefficiencies. We follow a top down approach in which we first characterize the overall behavior of the program in terms of its activities, e.g., computation, communication, synchronization, memory accesses, I/O operations. We then analyze the various code regions of the program, e.g., loops, routines, code statements, and the activities performed within each region.

The characterization of the performance properties and inefficiencies of the program is based on the definition of various criteria. In this section, we define the criteria that identify the dominant activities and the dominant code regions of the program. Next section is dedicated to the identification of inefficiencies due to dissimilarities in the behavior of the processors.

The performance of a parallel program is characterized by timings parameters, such as, wall clock times, as well as counting parameters, such as, number of I/O operations, number of bytes read/written, number of memory accesses, number of cache misses. Note that, not to clutter the presentation, in what follows we focus on timings parameters.

Let N denote the number of code regions of the parallel program, K the number of its activities, and P the number of allocated processors. t_{ijp} ($i = 1, 2, ..., N$; $j = 1, 2, ..., K$; $p = 1, 2, ..., P$) is the wall clock time of processor p in the activity j of the code region i. t_{ij} ($i = 1, 2, ..., N$; $j = 1, 2, ..., K$) is the wall clock time of the activity j in the code region i, that is:

$$t_{ij} = \frac{1}{P} \sum_{p=1}^{P} t_{ijp} \ .$$

Similarly, t_i ($i = 1, 2, ..., N$) is the wall clock time of the code region i, T_j ($j = 1, 2, ..., K$) is the wall clock time of the activity j, and T is the wall clock time of the whole program.

A preliminary characterization of the performance of a parallel program is based on the breakdown of its wall clock time T into the times T_j, ($j = 1, 2, ..., K$) spent in the various activities. The activity with the maximum T_j is defined as the dominant, that is, "heaviest", activity of the program, and could correspond to a performance bottleneck.

The analysis of the code regions is aimed at identifying the portions of the code where the program spends most of its time. The region with the maximum wall clock time, i.e., the heaviest region, might correspond to an inefficient portion of the program or to its core.

A refinement of this analysis is based on the breakdown of the wall clock time t_i into the times t_{ij} spent in the various activities. It might be difficult to understand which activity better explains the behavior and the performance of the program. We can identify the code region characterized by the maximum time in the dominant activity of the program. Moreover, for each activity j we can identify the worst and the best code regions, that is, with the maximum and minimum t_{ij}, respectively. This analysis results in a large amount of information. Hence, it is useful to summarize the properties of the program by identifying patterns or groups of regions characterized by a similar behavior. Clustering techniques [4] work for this purpose. Each code region i is described by its wall clock times t_{ij} and is represented in a K−dimensional space. Clustering partitions this space into groups of code regions with homogeneous characteristics such that the candidates for possible tuning are identified.

3 Processor Dissimilarities

The coarse grain analysis of the performance properties of parallel programs is followed by a fine grain analysis that focuses on the behavior of the processors with the objective of studying their load imbalance.
Load balancing is an ideal condition for a program to achieve good performance by fully exploiting the benefits of parallel computing. Programming inefficiencies might lead to uneven work distributions among processors. These distributions then lead to poor performance because of the delays due to loss of synchronization, dependencies and resource contentions among the processors.

Our methodology analyzes whether and where a program experienced poor performance because of load imbalance. For this purpose, we study the dissimilarities in the behavior of the processors with the aim of identifying the symptoms of uneven work distributions. In particular, we study the spread of the t_{ijp}'s, that is, the wall clock times spent by the various processors to perform activity j within code region i. As a first step, we need to define the metrics that detect and quantify dissimilarities and the criteria that assess their severity.

The metrics for evaluating the dissimilarities rely on the majorization theory [8], which provides a framework for measuring the spread of data sets. Such a theory is based on the definition of indices for partially ordering data sets according to the dissimilarities among their elements. The theory allows the identification of the data sets that are more spread out than the others. Dissimilarities can be measured by different indices of dispersion, such as, variance, coefficient of variation, Euclidean distance, mean absolute deviation, maximum, sum of the elements of the data sets. The choice of the most appropriate index of dispersion depends on the objective of the study and on the type of physical phenomenon to be analyzed. In our study, the index of dispersion has to measure the spread of the times spent by the processors to perform a given activity with respect to the perfectly balanced condition, where all processors spend exactly the same amount of time. The Euclidean distance between the time of each processor and the corresponding average is then well suited for our purpose.

Once the metrics to quantify dissimilarities have been defined, it is necessary to select the criteria for their ranking. The choice of the most appropriate criterion to assess the severity of the load imbalance among processors depends on the level of details required by the analysis. Possible criteria are the maximum of the indices of dispersion, the percentiles of their distribution, or some predefined thresholds.

The analysis of dissimilarities can then be summarized by the following steps:

- standardization of the wall clock times;
- computation of the indices of dispersion;
- ranking of the indices of dispersion.

Note that as the indices of dispersion have to provide a relative measure of the spread of the wall clock times, the first step of the methodology deals with a standardization of the wall clock times of each code region. As we will see, the standardized times are such that they sum to one, that is, they are obtained by dividing the wall clock times by the corresponding sum.

The second step of the methodology deals with the computation of the various indices of dispersion. In particular, our analysis focuses on three different views, namely, processor, activity, and code region. These views provide complementary insights into the behavior of the processors as they correspond to the different perspectives used to characterize a parallel program.

Once the indices of dispersion have been computed for the various views, their ranking allows us to identify processors, activities and code regions characterized by large dissimilarities which could be chosen as candidates for performance tuning.

3.1 Processor View

Processor view is aimed at analyzing the behavior of the processors across the activities performed within each code region with the objective of identifying the most frequently imbalanced processor. We describe the dissimilarities of each code region with P indices of dispersion ID_P_{ip}, one for each processor. These indices are computed as the Euclidean distance between the times spent by processor p on the various activities performed within code region i and the average time of these activities over all processors:

$$ID\_P_{ip} = \sqrt{\sum_{j=1}^{K} (\tilde{t}_{ijp} - \tilde{T}_{ij})^2} \ .$$

Note that the $\tilde{t}_{ijp}$'s are obtained by standardizing the t_{ijp}'s over the sum of the times spent by each processor in the various activities performed within a given code region. $\tilde{T}_{ij}$ denotes the corresponding average.

From the various indices of dispersion, we can identify the processors that have been most frequently imbalanced and imbalanced for the longest time.

3.2 Activity View

Activity view analyzes dissimilarities within the activities performed by the processors across all the code regions with the objective of identifying the most imbalanced activity. We first quantify the dissimilarities in the times spent by the various processors to perform a given activity within a code region. Let ID_{ij} be the index of dispersion computed as the Euclidean distance between the times spent by the various processors to perform activity j within code region i and their average. We then summarize the ID_{ij}'s to identify and localize the activity characterized by the largest load imbalance.

ID_A_j is the relative measure of the load imbalance within the activity j and is obtained as the weighted average of the ID_{ij}'s. The weights represent the fractions of the overall wall clock time accounted by activity j within code region i, that is, $\frac{t_{ij}}{T_j}$. As activities with large dissimilarities might have a negligible impact on the overall performance of the program because of their short wall clock time, we scale the index of dispersion ID_A_j according to the fraction of the program wall clock time accounted by the activity itself, namely:

$$SID\_A_j = \frac{T_j}{T} \ ID\_A_j \ .$$

The scaled indices of dispersion SID_A_j allow us to identify the activities characterized by large dissimilarities and accounting for a significant fraction of the wall clock time of the program.

3.3 Code region View

Code region view analyzes the dissimilarities with respect to the various activities performed by the processors within each region with the objective of identifying the most imbalanced region. The computation of the dissimilarities is based on the ID_{ij}'s defined in the activity view. ID_C_i is a relative measure of the load imbalance within code region i, and is obtained as the weighted average of the ID_{ij}'s with respect to $\frac{t_{ij}}{t_i}$, that is, the fraction of the wall clock time of the code region accounted by activity j. As in the activity view, we scale the index of dispersion ID_C_i with respect to the fraction of the program wall clock time accounted by code region i, i.e., $\frac{t_i}{T}$, and we obtain the scaled index SID_C_i.

4 Application Example

In this section we illustrate our methodology on the analysis of the performance inefficiencies of a message passing computational fluid dynamic code. We focus on an execution of the program on $P = 16$ processors of an IBM Sp2. The measurements refer to 7 code regions corresponding to the main loops of the program. Moreover, within each region, four activities have been measured, namely, computation, point-to-point communications (i.e., `MPI_SEND, MPI_RECV`), collective communications (i.e., `MPI_REDUCE, MPI_ALLTOALL`), and synchronizations among processors (i.e., `MPI_BARRIER`). In what follows, we identify the loops of the application with a number, from 1 to 7.

Table 1 presents the wall clock time of each loop with the corresponding breakdown into the wall clock times of its activities. By profiling the program, that is, by looking where the time is spent, we notice that the heaviest loop, that is, loop 1, accounts for about 27% of the overall wall clock time. This loop, which corresponds to the core of the program, is characterized by the longest time in computation, that is, the dominant activity of the program, as well as in collective communications and synchronizations, whereas it does not perform any point-to-point communication. The loop which spends the longest time in point-to-point communications is loop 3. Moreover, only three loops perform synchronizations.

For a more detailed analysis of the behavior of the loops we applied the *k-means* clustering algorithm [4]. Each loop is described the wall clock times it spent in the various activities. Clustering yields a partition of the loops into two groups. The heaviest loops of the program, that is, loops 1 and 2, belong to one group, whereas the remaining loops belong to the second group.

To gain better insights into the performance properties of the program and to study the dissimilarities in the processor behavior, we analyzed the wall clock times spent by the processors to perform the various activities. Figures 1 and 2 show the patterns of the times spent in computation and point-to-point communications activities, respectively. The patterns are plotted for each loop separately, namely, each row refers to one loop. Different colors are used to highlight the patterns.

Table 1. Overall wall clock time, in seconds, of the loops and corresponding breakdown

loop	wall clock time				
	overall	computation	point-to-point	collective	synchronization
1	19.051	12.24	-	6.75	0.061
2	14.22	7.90	-	6.32	-
3	10.90	5.22	5.68	-	-
4	10.54	8.03	2.51	-	-
5	9.041	7.53	0.07	1.43	0.011
6	0.692	0.36	0.33	-	0.002
7	0.31	0.28	-	0.03	-

The four colors used in the figures refer to the maximum and minimum values of the wall clock times of the loop and to values belonging to the lower and upper 15% intervals of the range of the wall clock times, respectively. Note that the diagrams plot only the loops performing the activity shown by the diagram itself.

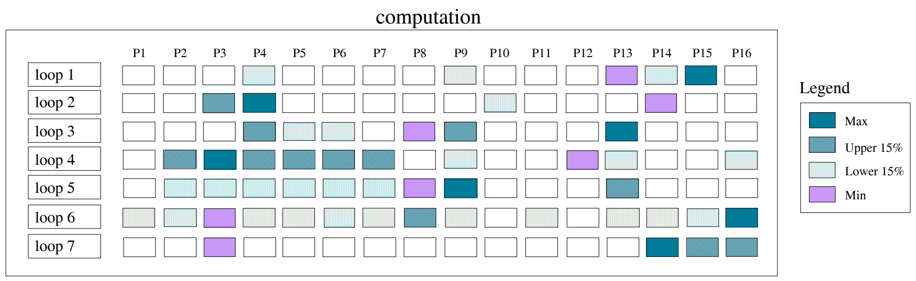

Fig. 1. Patterns of the times spent by the processors in computation

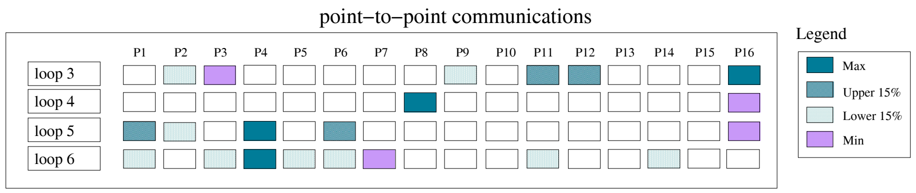

Fig. 2. Patterns of the times spent by the processors in point-to-point communications

As can be seen, the behavior of the processors within and across the various loops and activities is quite different. By analyzing the patterns shown in Figure 1, we notice that the times spent in computation by five out of 16 processors executing loop 4 belong to the upper 15% interval, whereas on loop 6 the times of 11 out of 16 processors belong to the lower 15% interval. From Figure 2 we can notice that the behavior of the processors executing point-to-point communications is very balanced.

These figures provide some qualitative insights into the behavior of the processors, whereas they lack in providing any quantitative description of their dissimilarities. To quantify the dissimilarities, we standardized the wall clock times and computed the indices of dispersion as defined in Section 3. From the analysis of the processor view, we have discovered that processor 1 is the most frequently imbalanced as it is characterized by the largest values of the index of dispersion on two loops, namely, loops 3 and 7. Processor 2 is imbalanced for the longest time. This processor is the most imbalanced on one loop only, namely, loop 1, with an index of dispersion equal to 0.25754 and a wall clock time equal to 15.93 seconds.

For the analysis of the activity and code region views, we have computed the indices of dispersion ID_{ij} presented in Table 2. As can be seen, the behavior of the processors is highly imbalanced when performing synchronizations. The value of the index of dispersion corresponding to loop 5 is equal to 0.30571. Loop 1 is the most imbalanced with respect to the times spent by the processors for performing collective communications, whereas loop 6 is characterized by the largest indices of dispersion in two activities, namely, computation and point-to-point communications.

Table 2. Indices of dispersion ID_{ij} of the activities performed by the loops

loop	computation	point-to-point	collective	synchronization
1	0.03674	-	0.06793	0.12870
2	0.01095	-	0.00318	-
3	0.00672	0.02833	-	-
4	0.01615	0.10742	-	-
5	0.00933	0.08872	0.04907	0.30571
6	0.05017	0.23200	-	0.16163
7	0.00719	-	0.01138	-

To summarize the values of Table 2 by taking into account the relative weights of the wall clock times of the activities and of the loops, we computed the weighted average of the ID_{ij}'s. Tables 3 and 4 present the values of the indices of dispersions ID_A_j and ID_C_i computed for the activities and the loops, respectively. The tables also present the indices SID_A_j and SID_C_i scaled with respect to the fraction of the wall clock time accounted by each activity or loop, respectively.

As can be seen from Table 3, the synchronization is the most imbalanced activity. However, as it accounts only for 0.1% of the wall clock time of the program, its impact on the overall performance is negligible. Hence, this activity does not seem a suitable candidate for tuning, as also denoted by the value of the scaled index of dispersion which is equal to 0.00016.

Table 3. Summary of the indices of dispersion of the activity view

activity	ID_A	SID_A
computation	0.01904	0.01132
point-to-point	0.05973	0.00734
collective	0.03781	0.00786
synchronization	0.15559	0.00016

Table 4. Summary of the indices of dispersion of the code region view

loop	ID_C	SID_C
1	0.04809	0.01311
2	0.00750	0.00152
3	0.01798	0.00280
4	0.03790	0.00571
5	0.01655	0.00214
6	0.13734	0.00135
7	0.00760	0.00003

From the analysis of the summaries presented in Table 4, we can conclude that loop 6 is the most imbalanced. The value of its index of dispersion is equal to 0.13734. However, as this loop accounts for a very short wall clock, the value of the corresponding scaled index of dispersion is equal to 0.00135 only.
These metrics help the users in deciding which loop is the best candidate for performance tuning. In our study loop 1 is a good candidate as it is the core of the program and it is also characterized by large values of both the index of dispersion and its scaled counterpart.

5 Conclusions

The analysis of performance inefficiencies of parallel programs is a challenging issue. Users do not want to browse too many diagrams or, even worse, to dig into the tracefiles collected during the execution of their programs. They expect from performance tools answers to their performance problems. Thereby, tools should do what expert programmers do when tuning their programs, that is, detect the presence of inefficiencies, localize them and assess their severity.

The identification and localization of the performance inefficiencies of parallel programs are preliminary steps towards an automatic performance analysis. The methodology presented in this paper is aimed at isolating inefficiencies and load imbalance within a program by analyzing performance measurements related to its execution. From the measurements we derive various metrics that guide users in the interpretation of the behavior and of the performance properties of their programs.

As a future work, we plan to define and test new criteria for the identification and localization of performance inefficiencies. Hence, we will analyze measurements collected on different parallel systems for a large variety of scientific programs [3]. Moreover, we plan to integrate our methodology into a performance tool.

References

1. M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. Medea: A Tool for Workload Characterization of Parallel Systems. *IEEE Parallel and Distributed Technology*, 3(4):72–80, 1995.
2. L. DeRose, Y. Zhang, and D.A. Reed. SvPablo: A Multi-Language Performance Analysis System. In R. Puigjaner, N. Savino, and B. Serra, editors, *Computer Performance Evaluation - Modelling Techniques and Tools*, volume 1469 of *Lecture Notes in Computer Science*, pages 352–355. Springer, 1998.
3. K. Ferschweiler, S. Harrah, D. Keon, M. Calzarossa, D. Tessera, and C. Pancake. The Tracefile Testbed – A Community Repository for Identifying and Retrieving HPC Performance Data. In *Proc. 2002 International Conference on Parallel Processing*, pages 177–184. IEEE Press, 2002.
4. J.A. Hartigan. *Clustering Algorithms.* Wiley, 1975.
5. M.T. Heath and J.A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8:29–39, 1991.
6. B. Helm, A. Malony, and S. Fickas. Capturing and Automating Performance Diagnosis: the Poirot Approach. In *Proceedings of the 1995 International Parallel Processing Symposium*, pages 606–613, 1995.
7. K.L. Karavanic and B.P. Miller. Improving Online Performance Diagnosis by the Use of Historical Performance Data. In *Proc. SC'99*, 1999.
8. A.W. Marshall and I. Olkin. *Inequalities: Theory of Majorization and Its Applications.* Academic Press, 1979.
9. B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K.H. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Measurement Performance Tool. *IEEE Computer*, 28(11):37–46, 1995.
10. P.C. Roth and B.P. Miller. Deep Start: A Hybrid Strategy for Automated Performance Problem Searches. In *Proc. 8th International Euro-Par Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 86–96. Springer, 2002.
11. M.L. Simmons, A.H. Hayes, J.S. Brown, and D.A. Reed, editors. *Debugging and Performance Tuning for Parallel Computing Systems.* IEEE Computer Society, 1996.
12. W. Williams, T. Hoel, and D. Pase. The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D. In K.M. Decker, editor, *Programming Environments for Massively Parallel Distributed Systems*, pages 333–345. Birkhauser Verlag, 1994.
13. J.C. Yan and S.R. Sarukkai. Analyzing Parallel Program Performance Using Normalized Performance Indices and Trace Transformation Techniques. *Parallel Computing*, 22(9):1215–1237, 1996.
14. O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward Scalable Performance Visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, 13(2):277–288, 1999.